escuelaingenieríaindustrial
PONTIFICIA UNIVERSIDAD CATOLICA DE VALPARAISO

Resolución de problemas a través de su implementación computacional: aspectos estructurales

Franco Guidi Polanco
Escuela de Ingeniería Industrial
Pontificia Universidad Católica de Valparaíso, Chile
fguidi@ucv.cl

Actualización: 02 de agosto de 2007

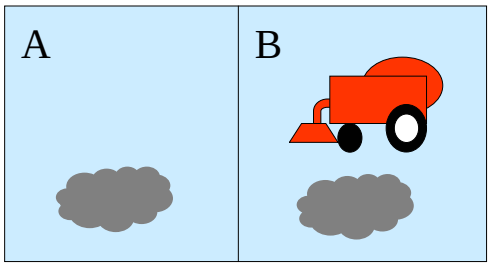
Resolución de problemas / construcción de sistemas

- Entender el problema y sus requerimientos
- Determinar un modelo que nos permita resolver el problema.
- Determinar la forma de representar el modelo que describe el problema (y la solución):

La solución debe considerar:
- Aspectos estructurales
- Aspectos funcionales
- Aspectos dinámicos

Ejemplo

- El “mundo de la aspiradora”



Dos posiciones A y B, inicialmente con polvo, y una aspiradora que parte en la posición B. La aspiradora aspira el polvo de la posición en que se encuentra.

Representación conceptual

- Mediante vectores o matrices booleanas:

$$P = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

1: presencia de polvo
0: ausencia de polvo

$$A = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

1: presencia de aspiradora
0: ausencia de aspiradora

Representación conceptual (cont.)

- Lógica proposicional

$$P_A \wedge P_B \wedge \neg A_A \wedge A_B$$

Donde:

P*i* : Presencia de polvo en posición *i*-ésima

A*i* : Presencia de aspiradora en posición *i*-ésima

Representación conceptual (cont.)

- Lógica de primer orden:

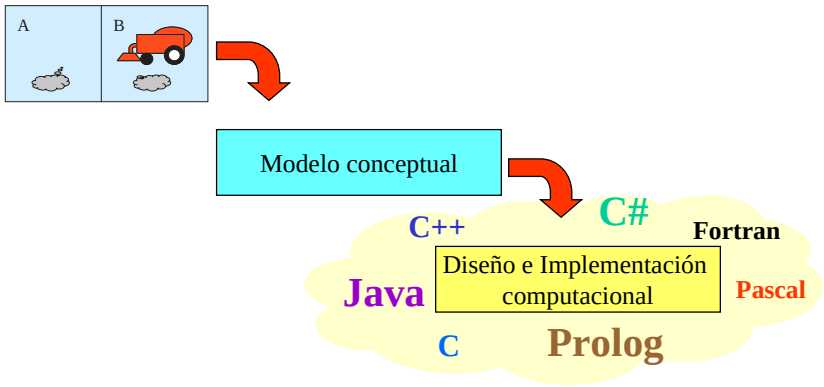
$$Polvo(A, t_1) \wedge Polvo(B, t_1) \wedge Aspiradora(B, A1, t_1)$$
$$Encendida(A1, t_1)$$

Reglas de inferencia:

$$\forall\, x, y, t_a, t_b \; Polvo(x, t_a) \wedge Aspiradora(x, y, t_a) \wedge$$
$$Encendida(y, t_a) \wedge Menor(t_a, t_b) \Rightarrow \neg Polvo(x, t_b)$$

Implementación computacional

- Varias opciones, dependiendo del modelo conceptual seleccionado, y del lenguaje de programación:



Implementación computacional (cont.)

- Ejemplo de implementación en Java:
  - Vectores implementados como arreglos booleanos independientes, para representar posiciones del polvo y la aspiradora:

```
boolean[] polvo = new boolean[2]
boolean[] aspiradora = new boolean[2]
```

  - Vectores implementados como una matriz de Java (en rigor arreglos de arreglos):

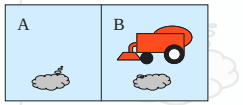
```
boolean[][] mundo = new boolean[2][2]
```

(Lectura: fila 0 = polvo; 1= aspiradora)

Implementación computacional (cont.)

- Ejemplo de implementación en Java:
  - Vector de polvo implementado como arreglos booleano, aspiradora como índice de posición :

```
boolean[] polvo;
int aspiradora;
```



=

```
polvo = { true; true }
aspiradora = 1;
```


Implementación computacional (cont.)

- Ejemplo de aplicación:

```
public class MiOficina {
   public static void main(String[] arg) {
     boolean polvo = new boolean[2];
     int aspiradora;
     …
     polvo[0] = true;
     polvo[1] = true;
     aspiradora = 1;
     …
     // Limpiar
     polvo[aspiradora] = false;
     System.out.println( "Sección "+
                         aspiradora + " limpia" );

     …
     if(aspiradora == 0)
          aspiradora = 1;
     else
          aspiradora = 0;
   }
}
```


¿Es conveniente la implementación anterior?

- A favor...

Implementación (aparentemente) bastante simple

- En contra...

Aplicación debe manejar detalles de la estructura
(Ej. subíndices)

Reutilización de código imposible o limitada
(Ej. Si deseo implementar la aplicación **`MiCasa`**)

Mejorando la implementación anterior: encapsulación

- Clase Mundo:

Podrían incluirse validaciones para las posiciones

```
public class Mundo {
        boolean[] polvo = new boolean[2];
        int aspiradora;
        public void setPolvo(int lugar){
                polvo[lugar] = true;
        }
        public void setAspiradora(int lugar){
                aspiradora = lugar;
        }
        public void limpiar(){
                polvo[aspiradora] = false;
        }
        public boolean estáLimpio(int i){
                return polvo[i];
        }
}
```

Mejorando la implementación anterior: encapsulación (cont.)

```
public class Oficina {
  public static void main(String[] arg) {
    Mundo mundo = new Mundo();
    …
    // Ensuciamos todo
    mundo.setPolvo( 0 );
    mundo.setPolvo( 1 );

    // Ubicamos la aspiradora
    mundo.setAspiradora( 1 );
    mundo.limpiar();
    …
    if( mundo.estáLimpio(0) && mundo.estáLimpio(1))
      System.out.println( "El mundo está totalmente limpio." );
    …
    if(aspiradora == 0) // Mueve aspiradora
      mundo.setAspiradora( 0 );
    else
      mundo.setAspiradora( 1 );
    mundo.limpiar();
…
   }
}
```


Comentarios sobre la implementación

- La clase Mundo...
  - ...Oculta bastantes detalles de implementación
  - ...Es reutilizable

...pero...

- "bastantes" puede no ser suficiente
  - La aplicación es todavía responsable de manejar la lógica del mundo (Ej.: al mover la aspiradora debe además invocar el método limpiar).
  - La aplicación accede a detalles de implementación (subíndices)

Nueva propuesta de implementación

- Clase Mundo

```
public class Mundo {
        boolean[] polvo = new boolean[2];
        int aspiradora;
        public static final LUGAR_A = 0;
        public static final LUGAR_B = 1;
        public void setPolvo(){
            polvo[LUGAR_A] = true; polvo[LUGAR_B] = true;
        }
        public void setAspiradora(int lugar){
                aspiradora = lugar;
                limpiar();
        }
        public void limpiar(){
                polvo[aspiradora] = false;
        }
        public boolean estáLimpio(){
                return polvo[LUGAR_A] && polvo[LUGAR_B];
        }
        public void moverAspiradora(){
                aspiradora = ( aspiradora == LUGAR_A?
                                LUGAR_B: LUGAR_A);
                limpiar();
        }
}
```


Nueva propuesta de implementación (cont.)

```
public class Oficina {
  public static void main(String[] arg) {
    Mundo mundo = new Mundo();
    …
    // Ensuciamos todo
    mundo.setPolvo();

    // Ubicamos la aspiradora
    mundo.setAspiradora( Mundo.LUGAR_B );
    …
    if( mundo.estáLimpio())
      System.out.println( "El mundo está totalmente limpio." );
    …
    // Mueve aspiradora
      mundo.moverAspiradora();
    …
   }
}
```

Los detalles de implementación están completamente ocultos.

Otro ejemplo: una pila

- Se desea manejar una pila de Solicitudes. Se propone la siguiente clase:

```
public class PilaSolicitudes {
   Solicitud[] solicitud;
   int posActual = 0;
   public PilaSolicitudes( int largo ){
        solicitud = new Solicitud[ largo ];
   }
   public void push(Solicitud s){
        solicitud[posActual] = s;
        posActual++;
   }
   public Solicitud pop(){
         posActual--;
        return solicitud[posActual];
   }
}
```

Nota:

Una pila es una colección que opera bajo la disciplina LIFO (Last In First Out)

Otro ejemplo: una pila (cont.)

- La pila de solicitudes es utilizada de la siguiente forma:

```
public class Ejemplo {
   public static void main(String[] arg) {
        PilaSolicitudes pila = new PilaSolicitudes( 100 );
        …
        Solicitud s = new Solicitud( 1 );
        pila.push( s );
        …
        s = new Solicitud( 4 );
        pila.push( s );
        …
        s = pila.pop();
        …
   }
}
```

Notar que la clase **`PilaSolicitud`** esconde a la aplicación los detalles de implementación…

…¿Es suficiente?

Otro ejemplo: una pila (cont.)

- Compare con la siguiente implementación de pila:

```
public class Pila {
   Object[] elemento;
   int posActual = 0;
   public Pila ( int largo ){
        elemento = new Object[ largo ];
   }
   public void push(Object s){
        elemento[posActual] = s;
        posActual++;
   }
   public Object pop(){
         posActual--;
        return elemento[posActual];
   }
}
```


Otro ejemplo: una pila (cont.)

- Características de ambas implementaciones:

`PilaSolicitudes`

- Reutilizable en el dominio del manejo de solicitudes (i.e. puede servir en el desarrollo de aplicaciones que utilicen la clase Solicitud)
- Trabaja directamente con objetos del dominio (ej. el método **`pop()`** retorna directamente una Solicitud)

`Pila`

- Reutilizable en cualquier aplicación que requiera manejar una pila.
- Trabaja con objetos genéricos (ej. el método **`pop()`** retorna un **`Object`**, que es necesario transformar a **`Solicitud`** en la aplicación)

Otro ejemplo: una pila (cont.)

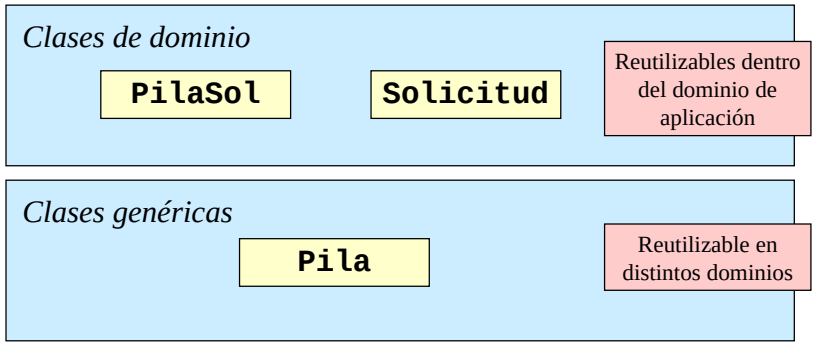
- Compatibilizando ventajas... Suponer que se mantiene la clase Pila y se crea la siguiente clase *wrapper*:

```
public class PilaSol {
   Pila pila;
   public PilaSol ( int largo ){
        pila = new Pila( largo );
   }
   public void push(Solicitud s){
        pila.push( s );
   }
   public Solicitud pop(){
        return ( (Solicitud) pila.pop() );
   }
}
```

La clase **`Pila`** sigue siendo reutilizable; la clase **`PilaSol`** ofrece una interfaz más adecuada para la aplicación.

Otro ejemplo: una pila (cont.)

- Reutilización y dominios:



Niveles de reutilización

- La reutilización puede lograrse considerando distintos niveles:
  - Clases → Librerías de clases
    - Clases utilizables en distintas aplicaciones
    - Ejemplo: clases para manejar Listas o Pilas.
  - Diseño → Patrones de diseño (*design patterns*)
    - Modelos arquitecturales que resuelven situaciones similares en dominios de aplicación distintos
    - Ejemplos: Adapter, Composite, Factory method.
  - Diseño y clases → Frameworks de software
    - Modelos arquitecturales e implementación parcial de funcionalidades para el desarrollo de aplicaciones en ciertos dominios
    - Ejemplos: framework de acceso a bases de datos, framework de sistemas multi-agente.
- El problema de la reutilización es estudiado dentro de la Ingeniería del Software

Resumen

- La solución de problemas a través del diseño de aplicaciones requiere:
  - La identificación del modelo conceptual del problema
  - Desarrollo de modelo de análisis y de diseño
  - Definición de su implementación computacional en términos de estructuras de datos y operaciones (clases)
- La implementación computacional se ve simplificada por la reutilización de software.
- La reutilización de software antes de la POO se basaba en la definición de estructuras de datos y de operaciones.
- La reutilización de software en la era de la POO se basa en el diseño de modelos y clases reutilizables (clases, patrones de diseño y *frameworks*)